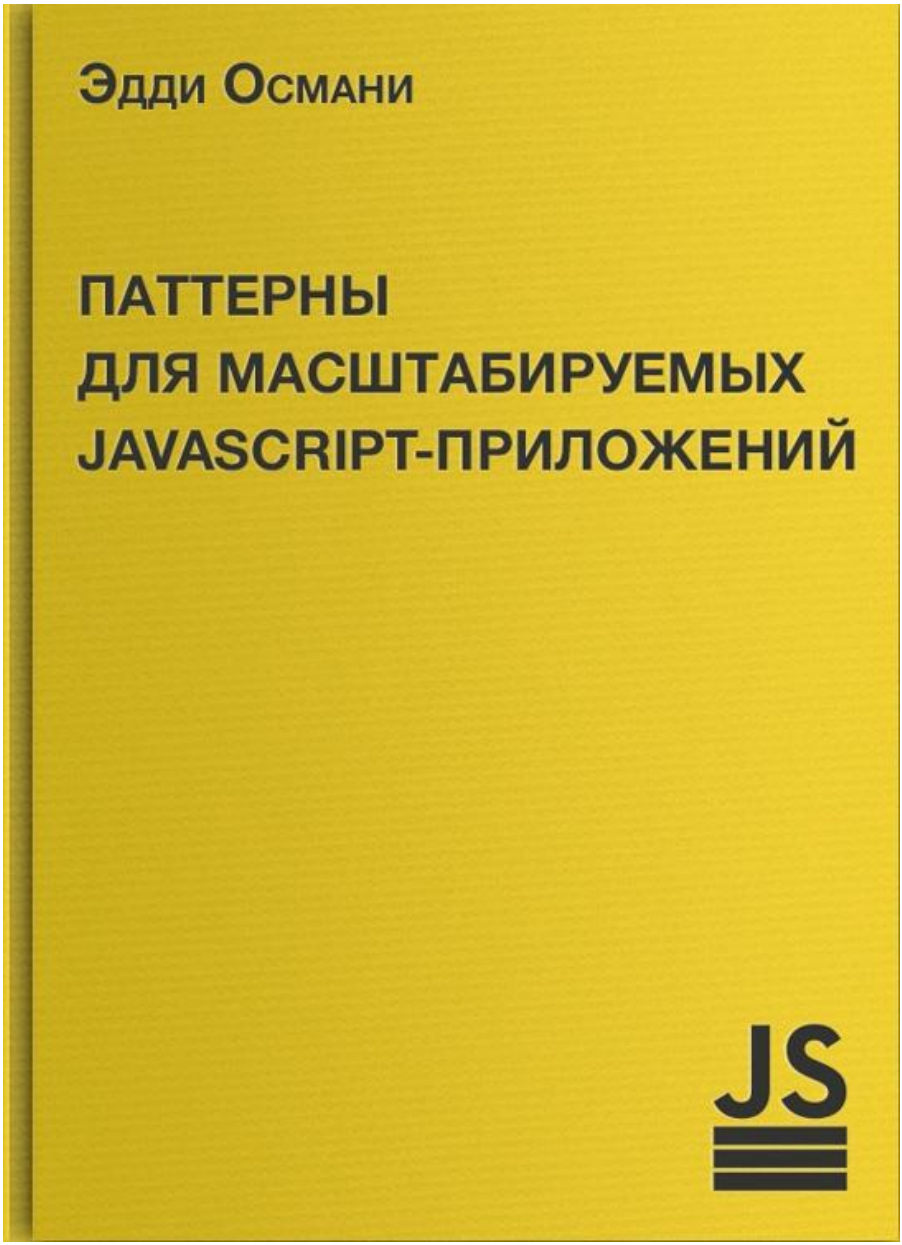
Эдди Османи

ПАТТЕРНЫ ДЛЯ МАСШТАБИРУЕМЫХ JAVASCRIPT-ПРИЛОЖЕНИЙ



Паттерны для масштабируемых

JavaScript-приложений

Паттерны для масштабируемых JavaScript-приложений

Вступление

Что из себя представляет «большое» JavaScript приложение?

Давайте обсудим вашу существующую архитектуру

Думай о будущем

Мозговой штурм

Теория модулей

Паттерн «Модуль»

Литеральная нотация объекта

CommonJS Модули

Паттерн «Фасад»

Паттерн «Медиатор»

Использование фасада: абстракция ядра

Использование медиатора: ядро приложения

Собираем всех вместе

Развитие идей медиатора: автоматическая регистрация событий

Frequently Asked Questions

Credits

Вступление

В этой книге мы обсудим набор паттернов, который поможет вам в создании

больших масштабируемых JavaScript-приложений. Материал книги основан на

моем одноименном докладе, впервые прочитанном на конференции

«LondonJS», и вдохновленном предшествующей ему работой Николаса Закаса.

Кто я и почему я решил об этом написать?

Меня зовут Эдди Османи. Сейчас я работаю JavaScript- и UI-разработчиком в

AOL. Я занимаюсь планированием и написанием фронтенд-архитектуры для

следующего поколения наших пользовательских приложений. Эти приложения

весьма сложны. Они нуждаются в архитектуре, позволяющей, с одной стороны

легко их масштабировать, а с другой достаточно легко использовать повторно

их модули. Также я занимаюсь разработкой шаблонов, которые можно

применять в разработке приложений подобного масштаба настолько

качественно, насколько это вообще возможно.

Кроме того, я рассматриваю себя как евангелиста шаблонов проектирования

(хотя есть много экспертов, разбирающихся в этом лучше меня). В прошлом я

написал книгу «Essential JavaScript Design Patterns», а сейчас я занимаюсь

написанием более подробного продолжения этой книги.

Могу ли я уместить эту книгу в 140 символов?

Я уместил эту статью в один твит, на случай, если у вас совсем мало времени: Меньше связанности: используйте паттерны «модуль», «фасад» и «медиатор».

Модули общаются через медиатор, а фасад обеспечивает безопасность.

Что из себя представляет «большое» JavaScript приложение?

Перед тем как я начну, давайте постараемся определить, что именно мы имеем

в виду, когда говорим о больших JavaScript-приложениях. Этот вопрос я считаю

своего рода вызовом опытным разработчикам, и ответы на него, соответственно, получаются очень субъективными.

Ради эксперимента я предложил нескольким среднестатистическим

разработчикам дать собственное определение этому термину. Один из

разработчиков сказал, что речь идет о «JavaScript-приложениях, состоящих из

более чем 100 000 строк кода», когда другой определил что большое

приложение «содержит больше чем 1 МБ JavaScript». Я расстроил храбрецов —

оба этих варианта далеки от истины. Количество кода не всегда коррелирует со

сложностью приложения. 100 000 строк легко могут оказаться самым ничем не

примечательным простым кодом.

Я не знаю, подходит ли мое собственное определение к любому случаю, но я

верю, что оно находится ближе всего к тому, что действительно представляет

из себя большое JavaScript-приложение.

Я думаю, что большие JavaScript-приложения решают нетривиальные задачи, а

поддержка таких приложений требует от разработчика серьезных усилий. При

этом, большая часть работы по манипуляции данными и их отображению

ложится на браузер.

Я думаю, что последняя часть определения — самая важная.

Давайте обсудим вашу существующую архитектуру

Работая над большим JavaScript-приложением, не забывайте уделять

достаточное количество времени на планирование изначальной архитектуры, к которой такие приложения очень чувствительны. Большие приложения

обычно представляют из себя очень сложные системы, гораздо более сложные, чем вы представляете себе изначально.

Я должен подчеркнуть значение этой разницы — я видел разработчиков, которые, сталкиваясь с большими приложениями, делали шаг назад и говорили:

«Хорошо, у меня есть несколько идей, которые хорошо показали себя в моем

предыдущем проекте среднего масштаба. Думаю, они точно сработают и для

чего-то большего, не так ли?». Конечно, до какого-то момента это может быть

так, но, пожалуйста, не принимайте это как должное — по большей части

большие приложения имеют ряд достаточно серьезных проблем, с которыми

нужно считаться. Ниже я приведу несколько доводов в пользу того, почему вам

стоит уделить немного больше времени планированию архитектуры своего

приложения, и чем вам это будет полезно в долгосрочной перспективе.

Большинство JavaScript-разработчиков в архитектуре своих приложений

обычно использует различные комбинации следующих компонентов: виджеты

модели

представления

контроллеры

шаблоны

библиотеки

ядро приложения.

Ссылки по теме:

Ребекка Мёрфи — Создание архитектуры JavaScript-приложений

Питер Мишо — MVC архитектура для JavaScript-приложений

StackOverflow — Дискуссия о современных MVC-фреймворках

Дуг Найнер — Поддерживаемые плагины и фабрика виджетов

Вероятно, вы выносите различные функции ваших приложений в отдельные

модули, либо используете какие-нибудь другие шаблоны проектирования для

подобного разделения. Это очень хорошо, но здесь есть ряд потенциальных

проблем, с которыми вы можете столкнуться при таком подходе.

1. Готова ли ваша архитектура к повторному использованию

кода уже сейчас?

Могут ли отдельные модули использоваться самостоятельно? Достаточно ли

они автономны для этого? Мог бы я прямо сейчас взять один из модулей вашего

большого приложения, просто поместить его на новую веб-страницу, а затем, тут же, начать его использовать? У вас может возникнуть вопрос:

«Действительно ли это так необходимо?», но, как бы то ни было, я

надеюсь, что

вы думаете о будущем. Что, если ваша компания начнет создавать все больше и

больше нетривиальных приложений, которые будут иметь некоторую общую

функциональность? Если кто-то скажет: «Нашим пользователям очень

нравится использовать модуль чата в нашем email-клиенте, почему бы нам не

добавить этот модуль к нашему новому приложению для совместной работы с

документами?», будет ли это возможно, если мы не уделим должного внимания

контролю кода?

2. Сколько модулей в вашей системе зависит от других модулей?

Насколько сильно связаны ваши модули? Перед тем, как я погружусь в

объяснения о том, как важна слабая связанность модулей, я должен отметить, что не всегда есть возможность создавать модули, не имеющие абсолютно

никаких зависимостей в системе. К примеру, одни модули могут расширять

функции других, уже существующих. Эта тема, скорее всего, относится к

группировке модулей на основе некоторой функциональности. Отдельные

наборы модулей должны работать в вашем приложении без большого

количества зависимостей, чтобы наличие или загрузка других модулей не

влияла на их работоспособность.

3. Сможет ли ваше приложение работать дальше, если его

отдельная часть сломается?

Если вы разрабатываете приложения, подобные Gmail, и ваш webmail-модуль

(или группа модулей) перестанет работать из-за ошибки, то это не должно

заблокировать пользовательский интерфейс или помешать пользователям

использовать другие части вашего приложения, к примеру, такие как чат. В то

же время, как было сказано раньше, было бы идеально, если бы модули могли

работать и за пределами вашей архитектуры. В моей лекции я упоминал

динамические зависимости — возможность загружать модули, исходя из

определенных действий пользователя. К примеру, ребята из Gmail могли бы

изначально держать чат закрытым, не загружая его код при открытии

страницы. А в тот момент, когда пользователь решит воспользоваться им —

соответствующий модуль будет динамически загружен и выполнен. В

идеальном случае хотелось бы выполнить это без каких-то негативных

эффектов в вашем приложении.

4. Насколько легко вы сможете тестировать отдельные модули?

Когда вы работаете над масштабными системами, есть вероятность, что

различные части этой системы будут использовать миллионы пользователей.

Вполне вероятно, что эти части будут использоваться не только в

предусмотренных вами ситуациях. В конечном счете, код может

использоваться повторно в огромном количестве различных окружений, и

важно, чтобы модули были достаточно протестированы. Тестировать модули

необходимо и внутри архитектуры, для которой он был изначально разработан, и снаружи. По моему мнению, это дает наибольшую гарантию того, что модуль

не сломается при попадании в другую систему.

Думай о будущем

В процессе создания архитектуры большого приложения, очень важно думать о

будущем. Не только о том, что будет через месяц или через год, но и о том, что

будет после этого. Что может измениться? Конечно, невозможно достаточно

точно предсказать как ваше приложение будет развиваться, но, вне всякого

сомнения, имеет смысл подумать об этом. Думаю, что найдется, хотя бы один

специфичный аспект вашего приложения, о котором стоит поразмыслить.

Разработчики зачастую слишком сильно связывают манипуляцию с DOM-элементами и остальные части приложения, даже если до этого они не

поленились разделить бизнес-логику на модули. Подумайте, почему в

долгосрочной перспективе это может быть плохой идеей?

Один из слушателей моей лекции предположил, что такая архитектура негибка, и может не работать в будущем. Это действительно так, но есть другая

проблема, игнорирование которой окажет еще более негативный эффект.

В будущем, вы можете принять решение о замене Dojo, jQuery, Zepto или YUI на что-нибудь совершенно иное. Причиной такого перехода может быть

производительность, безопасность или дизайн. Это может стать серьезной

проблемой, потому как библиотеки не предусматривают простой замены. Цена

замены библиотеки будет высокой, если ваше приложение тесно с ней связано.

Если вы используете Dojo (как многие слушатели на моей лекции), вы можете

быть уверены, что нет ничего лучше, на что имело бы смысл сейчас перейти.

Но можете ли вы быть уверены, что в течении двух-трех лет не появится что-нибудь, что вызовет у вас интерес? Что-нибудь, на что вы можете решить

перейти.

Такое решение может быть достаточно простым для небольших проектов, но не

для больших приложений, архитектура которых, в целом, достаточна их

поддержки.

Такое решение может быть достаточно простым для небольших проектов, но на

больших приложениях с непродуманной архитектурой всю тяжесть смены

библиотеки вы ощутите сразу.

Подводя итог, взгляните на вашу архитектуру сейчас. Сможете ли вы сегодня

сменить вашу библиотеку на любую другую, не переписывая при этом ваше

приложение полностью? Если это не так, то вам стоит продолжить чтение. Я

думаю, что в архитектуре, которую мы обсуждаем, вы найдёте кое-что

интересное.

Некоторые из известных JavaScript-разработчиков раньше уже излагали

проблемы, о которых я написал выше. Вот три ключевых цитаты, которыми я

бы хотел поделиться с вами.

«Секрет создания больших приложений в том, чтобы никогда не создавать

больших приложений. Разбейте ваши приложения на маленькие части, а затем

собирайте из этих маленьких тестируемых фрагментов ваше большое

приложение» Джастин Майер, автор «JavaScriptMVC»

«Секрет в том, чтобы признаться самому себе с самого начала, что вы понятия

не имеете о том, как ваше приложение будет развиваться. Когда вы согласитесь

с этим, вы начнете проектировать систему основываясь на защите. Вы

определите ключевые области, в которых, вероятнее всего будут происходить

изменения. Очень часто это не составляет труда, если потратить на это немного

времени. К примеру, вы ожидаете, что любая часть приложения, которая

взаимодействует с другой системой — это потенциальная мишень для

изменений. И вы понимаете, что здесь вам понадобится абстракция».

Николас Закас, автор книги «Высокопроизводительный JavaScript
И последняя, но тоже очень важная цитата:

«Чем сильнее компоненты связаны между собой, тем меньше возможностей

для их повторного использования, тем сложнее вносить изменения, не получая

при этом различных побочных эффектов в самых неожиданных местах»

Ребекка Мёрфи, автор книги «Фундаментальные основы jQuery»

Эти принципы необходимы для создания архитектуры, способной выдержать

испытание временем. Важно всегда помнить о них.

Мозговой штурм

Давайте немного подумаем, что мы хотим получить.

Мы хотим получить слабосвязанную архитектуру с функциональностью, разделенную на независимые модули, которые, в идеале, не должны иметь

зависимостей друг от друга. Когда случается что-то интересное, модули

сообщают об этом другим частям приложения, а промежуточный слой

интерпретирует их сообщения и необходимым образом реагирует на них.

Для примера, у нас есть JavaScript-приложение, отвечающее за онлайн-пекарню. Одно из интересных нам сообщений может быть

таким: «Партия из

42 батонов готова к доставке».

Мы используем отдельный слой для обработки сообщений модулей, чтобы а) модули не взаимодействовали напрямую с ядром, б) модули не

взаимодействовали напрямую друг с другом. Это помогает не допустить

падения приложения из-за различных ошибок внутри одного из модулей. Также

это позволяет нам перезапускать модули если они вдруг перестали работать из-за какой-нибудь ошибки.

Еще один момент — безопасность. На самом деле, немногие из нас заботятся о

внутренней безопасности своих приложений в должной мере. Когда мы

определяем структуру приложения, мы говорим себе, что мы достаточно умны

для того, чтобы понимать что в нашем коде должно быть публичным, а что

приватным.

Хорошо, но поможет ли это если вы решите определить что именно разрешено

модулю выполнять в системе? К примеру, в моем приложении, ограничив

доступ из модуля веб-чата к интерфейсу модуля администрирования, я смогу

уменьшить шансы на успешное использование XSS уязвимостей, которые я не

смог найти в виджете. Модули не должны иметь доступ ко всему. Вероятно, в

вашей существующей архитектуре они могут использовать любые

части

системы, но уверены ли вы, что это действительно необходимо?

Промежуточный слой, проверяющий имеет ли модуль доступ к определенной

части вашего фреймворка, обеспечивает большую безопасность вашей системы.

Фактически, это значит что модули могут взаимодействовать только с теми

компонентами системы, с которыми мы разрешим им взаимодействовать.

Архитектура, которую я предлагаю вам

Архитектура, о которой мы говорим, представляет из себя комбинацию трех

известных шаблонов проектирования: модуль, фасад и медиатор.

В отличии от традиционной модели, в которой модули напрямую

взаимодействуют друг с другом, в этой слабосвязанной архитектуре модули

всего лишь публикуют события (в идеале, не зная о других модулях в системе).

Медиатор используется для подписки на сообщения от модулей и для решения, каким должен быть ответ на уведомление. Паттерн фасад используется для

ограничения действий разрешенных модулям.

В следующих главах я более детально расскажу о каждом из этих шаблонов

проектирования.

Теория модулей

Вероятно, в каком-то виде вы уже используете модули в своей существующей

архитектуре. Если это не так, то в этой главе я покажу вам, как они устроены.

Модули — это целая часть любой хорошей архитектуры приложения. Обычно

модули выполняют одну определенную задачу в более крупных системах и

могут быть взаимозаменяемы.

В некоторых реализациях модули могут иметь свои собственные зависимости, которые будут загружены автоматически, собирая вместе таким образом все

компоненты системы. Такой подход считается более масштабируемым, в

отличие от ручной загрузки модулей или подстановки тега script.

Каждое нетривиальное приложение должно создаваться из модульных

компонентов. Рассмотрим GMail: вы можете рассматривать модули, как

независимые единицы функциональности, которые могут и должны

существовать сами по себе; возьмём к примеру чат. Скорее всего он основан на

своём отдельном модуле чата, но так как этот модуль скорее всего очень

сложный, то он вероятно состоит из более мелких вспомогательных модулей.

Например, один из таких модулей мог бы отвечать за использование смайликов

и он же мог бы использоваться не только в чате, но также и в почте.

В рассматриваемой архитектуре модули имеют очень ограниченные знания о

том, что происходит в других частях системы. Вместо этого мы делегируем

ответственность медиатору и фасаду.

В этом и заключается идея нашей архитектуры — если модуль заботится

исключительно о том, чтобы уведомить систему об интересующих ее

происшествиях, и не волнуется запущены ли другие модули, то система может

добавлять, удалять или заменять одни модули, не ломая при этом другие, что

было бы невозможно при сильной связанности.

Слабая связанность — необходимое условие для того, чтобы такая идея была

возможна. Она делает поддержку модулей проще, удаляя зависимости в коде

там, где это возможно. В вашем случае, одни модули должны работать

корректно в не зависимости от того в каком порядке загрузились другие

модули. Когда слабая связанность реализована эффективно, становится

очевидно, как изменения в одной части системы влияют на другие ее части.

В JavaScript есть несколько мнений о том, как могут быть реализованы модули,

включая шаблон «Модуль» и Object Literal (литеральная запись объекта

var obj = {};). Опытные разработчики должно быть уже знакомы с ними.

Если это так, то вы можете пропустите следующую главу и

перейти сразу к

главе «CommonJS Modules».

Паттерн «Модуль»

«Модуль» — это популярная реализация паттерна, инкапсулирующего

приватную информацию, состояние и структуру, используя замыкания. Это

позволяет оборачивать публичные и приватные методы и переменные в модули, и предотвращать их попадание в глобальный контекст, где они могут

конфликтовать с интерфейсами других разработчиков. Паттерн «модуль»

возвращает только публичную часть API, оставляя всё остальное доступным

только внутри замыканий.

Это хорошее решение для того, чтобы скрыть внутреннюю логику от

посторонних глаз и производить всю тяжелую работу исключительно через

интерфейс, который вы определите для использования в других частях вашего

приложения. Этот паттерн очень похож на немедленно-вызываемые функции

(IIFE), за тем исключением, что модуль вместо функции, возвращает объект.

Важно заметить, что в JavaScript нет настоящей приватности. В отличии от

некоторых традиционных языков, он не имеет модификаторов доступа.

Переменные технически не могут быть объявлены как публичные или

приватные, и нам приходится использовать область видимости для того, чтобы

эмулировать эту концепцию. Благодаря замыканию, объявленные внутри

модуля переменные и методы доступны только изнутри этого модуля.

Переменные и методы, объявленные внутри объекта, возвращаемого модулем, будут доступны всем.

Ниже вы можете увидеть корзину покупок, реализованную с помощью

паттерна «модуль». Получившийся компонент находится в глобальном объекте

basketModule, и содержит всё, что ему необходимо. Находящийся внутри него, массив basket приватный, и другие части вашего приложения не могут

напрямую взаимодействовать с ним. Массив basket существует внутри

замыкания, созданного модулем, и взаимодействовать с ним могут только

методы, находящиеся в том же контексте (например, addItem(), getItem()).

```
var basketModule = (function() {

var basket = []; // приватная переменная

return { // методы доступные извне

addItem: function(values) {

basket.push(values);

},

getItemCount: function() {

return basket.length;
```

```
},

getTotal: function() {

var q = this.getItemCount(),p=0; while(q--){

p+= basket[q].price;

}

return p;

}

}

}());
```

Внутри модуля, как вы заметили, мы возвращаем объект. Этот объект

автоматически присваивается переменной basketModule, так что с ним можно

взаимодействовать следующим образом:

```
// basketModule - это объект со свойствами, которые могут также быть и методами: basketModule.addItem({item:'bread', price:0.5});

basketModule.addItem({item:'butter', price:0.3});

console.log(basketModule.getItemCount());

console.log(basketModule.getTotal());

// А следующий ниже код работать не будет:

console.log(basketModule.basket); // undefined потому что не входит в возвращаемый объект

console.log(basket); // массив доступен только из замыкания
```

Методы выше, фактически, помещены в неймспейс basketModule.

Исторически, паттерн «модуль» был разработан в 2003 году группой людей, в

число которых входил Ричард Корнфорд. Позднее, этот паттерн

был

популяризован Дугласом Крокфордом в его лекциях, и открыт заново в блоге

YUI благодаря Эрику Мирагилиа.

Давайте посмотрим на реализацию «модуля» в различных библиотеках и

фреймворках.

Dojo

Dojo старается обеспечивать поведение похожее на классы с помощью

dojo.declare, который, кроме создания «модулей», также используется и для

других вещей. Давайте попробуем, для примера, определить basket как модуль

внутри неймспейса store:

```
// традиционный способ

var store = window.store || {};

store.basket = store.basket || {};

// с помощью dojo.setObject

dojo.setObject("store.basket.object", (function() {

var basket = [];

function privateMethod() {

console.log(basket);

}

return {

publicMethod: function() {

privateMethod();
```

```
}

};

}()));
```

Лучшего результата можно добиться, используя dojo.provide и миксины.

YUI

Следующий код, по большей части, основан на примере реализации паттерна

«модуль» в фреймворке YUI, разработанным Эриком Миргалиа, но более

самодокументирован.

```
YAHOO.store.basket = function () {

// приватная переменная:

var myPrivateVar = "Ко мне можно получить доступ только из YAHOO.store.basket.";


// приватный метод:

var myPrivateMethod = function() {

YAHOO.log("Я доступен только при вызове из YAHOO.store.basket");

}


return {

myPublicProperty: "Я - публичное свойство", myPublicMethod: function() {

YAHOO.log("Я - публичный метод");


// Будучи внутри корзины я могу получить доступ к приватным переменный и методам: YAHOO.log(myPrivateVar);
```

```
YAHOO.log(myPrivateMethod());


// Родной контекст метода myPublicMethod сохранён

// поэтому мы имеет доступ к this

YAHOO.log(this.myPublicProperty);

}

};


}();
```

jQuery

Существует множество способов, чтобы представить jQuery-код в виде

паттерна «модуль», даже если этот код не напоминает привычные jQuery-плагины. Бен Черри ранее предлагал способ, при котором, если у модулей есть

общие черты, то они объявляются через функцию-обертку.

В следующем примере функция library используется для объявления новой

библиотеки и, автоматически, при создании библиотеки (т.е. модуля), связывает вызов метода init с document.ready.

```
function library(module) {

$(function() {

if (module.init) {

module.init();

}

});

return module;

}
```

```
var myLibrary = library(function() {

return {

init: function() {

/* код модуля */

}

};

}());
```

Ссылки по теме:

Бен Черри — Погружение в паттерн «Модуль»

Джон Ханн — Будущее — это модули, а не фреймворки

Натан Смит — Ссылки на window и document в модулях (gist)

Литеральная нотация объекта

В литеральной нотации объект описывается внутри блока фигурных скобок

({}), как набор разделенных запятой пар ключ/значение. Ключи объекта могут

быть как строками, так и идентификаторами. После имени ставится двоеточие.

В объекте не должно стоять запятой после последней пары ключ/значение, так

как это может привести к ошибкам.

Литерал объекта не требует использования оператора new для создания

экземпляра, но он не должен стоять в начале выражения, так как открытая {

может быть воспринята как начало блока. Ниже вы можете увидеть пример

модуля, определенного с помощью литеральной нотации объекта.

Новые члены

объекта могут быть добавлены с помощью конструкции myModule.property =

'someValue';

Паттерн «модуль» может быть полезен для многих вещей. Но если вы считаете, что вам не нужно делать приватными некоторые методы или свойства, то

литерал объекта — более чем подходящий выбор.

var myModule = {

myProperty: 'someValue',

// Литералы объектов могут содержать свойства и методы.

// ниже в свойстве определен другой объект,

// для описания конфигурации:

myConfig: {

useCaching: true,

language: 'en'

},

// Очень простой метод

myMethod: function() {

console.log('I can haz functionality?');

},

// вывод значения заданного в конфигурации

myMethod2: function() {

console.log('Caching is: ' + ((this.myConfig.useCaching) ? 'enabled' : 'disabled'));

},

```
// переопределение конфигурации

myMethod3: function(newConfig) {

if (typeof newConfig == 'object') {

this.myConfig = newConfig;

console.log(this.myConfig.language);

}

}

};

myModule.myMethod(); // 'I can haz functionality'

myModule.myMethod2(); // Вывод 'enabled'

myModule.myMethod3({language:'fr',useCaching:false}); // 'fr'
```

Ссылки по теме:

Ребекка Мёрфи — Использование объектов для организации вашего кода

Стоян Стефанов — 3 способа определения класса в JavaScript

Бен Алман — Разъяснения по литералам объектов (понятия JSON-объект не

существует)

Джон Резиг - Простое наследование в JavaScript

CommonJS Модули

Возможно, вы что-то слышали о CommonJS за последние пару лет. CommonJS

— это добровольная рабочая группа, которая проектирует, прототипирует и

стандартизирует различные JavaScript API. На сегодняшний день они

ратифицировали стандарты для модулей и пакетов — CommonJS

определяют

простой API для написания модулей, которые могут быть использованы в

браузере с помощью тега <script>, как с синхронной, так и с асинхронной

загрузкой. Реализация паттерна «модуль» с помощью CommonJS выглядит

очень просто, и я нахожу это уверенным шагом на пути к модульной системе, предложенной в ES Harmony (следующей версии JavaScript).

В структурном плане, CommonJS-модуль представляет собой готовый к

переиспользованию

фрагмент

JavaScript-кода,

который

экспортирует

специальные объекты, доступные для использования в любом зависимом коде.

CommonJS все чаще используется как стандартный формат JavaScript-модулей.

Существует большое количество хороших уроков по написанию CommonJS-модулей, но обычно они описывают две главных идеи: объект exports, содержащий то, что модуль хочет сделать доступным для других частей

системы, и функцию require, которая используется одними модулями для

импорта объекта exports из других.

/*

Пример обеспечения совместимости между AMD и обычным

CommonJS с помощью

создания обертки над последним:

*/

(function(define) {

define(function(require,exports) {

// module contents

var dep1 = require("dep1");

exports.someExportedFunction = function() {...};

//...

});

})(typeof define==="function"?define:function(factory) {factory(require,exports)}); Есть много хороших JavaScript-библиотек, для загрузки модулей в формате

CommonJS, но моим личным предпочтением является RequireJS. Полный

учебник по RequireJS выходит за рамки этого руководства, но я могу

порекомендовать вам почитать пост Джеймса Брука «ScriptJunkie». Кроме того, я знаю многих людей, которые предпочитают Yabble.

Из коробки, RequireJS уже содержит методы для простого создания статичных

модулей с обертками. Благодаря этим методам становится действительно легко

создавать модули с поддержкой асинхронной загрузки. RequireJS может легко

загружать модули и их зависимости, выполняя тело модуля, сразу, как только

это становится возможным.

Некоторые разработчики утверждают, что CommonJS-модули не достаточно

удобны для применения в браузере, потому как без определенной помощи со

стороны сервера, их нельзя загрузить с помощью тега script. Давайте

представим, что есть некая библиотека для кодирования изображений в виде

ASCII-изображений, которая экспортирует функцию encodeToASCII. Модуль

использующий эту библиотеку будет выглядеть примерно так: var encodeToASCII = require("encoder").encodeToASCII; exports.encodeSomeSource = function() {

// Обработка изображения, затем вызов encodeToASCII

}

Этот код не будет работать с тегом script. Ему необходим определенный

контекст. Я имею в виду наш метод encodeToASCII, который ссылается на

несуществующие в контексте window методы require и exports. В такой

ситуации нам пришлось бы писать require и exports для каждого отдельного

модуля. Эту проблему легко решают клиентские библиотеки, которые

загружают скрипты через XHR-запросы, а затем выполняют eval().

Попробуем переписать этот модуль, используя RequireJS: define(function(require, exports, module) {

var encodeToASCII = require("encoder").encodeToASCII; exports.encodeSomeSource = function() {

// Обработка изображения, затем вызов encodeToASCII

}

});

Для разработчиков, которые хотят пойти дальше простого использования

JavaScript в своих проектах, CommonJS модули — прекрасная возможность

начать движение в эту сторону, но придется потратить немного времени и

познакомиться поближе с этим форматом. Все, что я рассказал — это только

верхушка айсберга. К счастью, CommonJS wiki и SitePen содержат много

материалов, которые помогут вам глубже разобраться в устройстве CommonJS-модулей.

Ссылки по теме:

Спецификации CommonJS-модулей

Алекс Янг - Прояснение CommonJS-модулей

Заметки о CommonJS- и RequireJS-модулях

Паттерн «Фасад»

Ключевую роль в архитектуре, которую мы обсуждаем в этой книге, играет

шаблон проектирования под названием «фасад».

Как правило, фасад используется для создания некоторой абстракции, скрывающей за собой совершенно иную реальность. Паттерн «фасад»

обеспечивает удобный высокоуровневый интерфейс для больших блоков

кода, скрывая за собой их истинную сложность. Относитесь к фасаду, как к

упрощенному API, который вы отдаете в пользование другим разработчикам.

Фасад — структурный паттерн. Часто его можно обнаружить в JavaScript-библиотеках и фреймворках, где пользователям доступнен только фасад —

ограниченная абстракция широкого диапазона поведений реализованных

внутри.

Благодаря такому подходу, пользователь взаимодействует только с

интерфейсом, не имея никакого представления о подсистемах, которые

скрываются за ним.

Причина, по которой нам интересен фасад — возможность скрыть детали

реализации конкретной функциональности, хранящиеся в модулях. Это

позволит нам вносить изменения в реализацию, не сообщая об этом

пользователям.

Надежный фасад — наш упрощенный интерфейс — позволит нам не

беспокоиться о тесных связях некоторых модулей нашей системы с dojo, jQuery, YUI, zepto или какой-либо другой библиотекой. Это становится не так

важно. Вы можете переходить с одной библиотеки на другую не меняя слой

взаимодействия. К примеру, с jQuery на dojo. Более того, у вас появляется

возможность совершить такой переход на более поздних этапах, без изменений

в остальных частях системы.

Ниже я написал достаточно простой пример использования фасада. Как вы

видите, у нашего модуля есть несколько приватных методов. Чтобы создать

более простой интерфейс для доступа к этим методам мы используем фасад.

```
var module = (function() {

var _private = {

i: 5,

get: function() {

console.log('Текущее значение:' + this.i);

},

set: function(val) {

this.i = val;

},

run: function() {

console.log('процесс запущен');

},

jump: function() {

console.log('резкое изменение');

}

};

return {

facade: function(args) {

_private.set(args.val);
```

```
_private.get();

if (args.run) {

_private.run();

}

}

}

}());

module.facade({run:true, val:10}); // Текущее значение: 10, процесс запущен
```

Это и есть та причина, по которой мы добавили фасад к нашей архитектуре. В

следующей главе мы обсудим медиатор. Принципиальное различие между

этими двумя паттернами заключается в том, что фасад, как структурный

паттерн, всего лишь передает существующую функциональность в медиатор, в

то время как медиатор, как поведенческий паттерн, может эту

функциональность расширять.

Ссылки по теме:

Дастин Диас, Росс Хармс — «Профессиональные шаблоны проектирования

JavaScript» (Глава 10, доступно для чтения в Google Books)

Паттерн «Медиатор»

Объяснить, что представляет собой паттерн «медиатор» достаточно просто на

примере следующей аналогии — представьте себе контроль траффика в

аэропорте: все решения о том, какие самолеты могут взлетать или садиться, принимает диспетчер. Для этого, все сообщения, исходящие от самолетов, поступают в башню управления, вместо того, чтобы пересылаться между

самолетами напрямую. Такой централизованный контроллер — это и есть

ключ к успеху нашей системы. Это и есть «медиатор».

Медиатор применяется в системах, где взаимодействие между модулями может

быть весьма сложными, но, в то же время, хорошо определенными. Если вы

полагаете, что связи между модулями вашей системы будут постоянно расти и

усложняться, то, возможно, вам стоит добавить центральный элемент

управления. Паттерн «медиатор» отлично подходит для этой роли.

Медиатор выступает в качестве посредника в общении между различными

модулями, инкапсулируя их взаимодействие. Кроме того, этот шаблон

проектирования,

предотвращая

прямое

взаимодействие

различных

компонентов системы, способствует ослаблению связей в коде. В нашей

системе он так же помогает в решении проблем, связанных с зависимостями

модулей.

Какие еще преимущества существуют у «медиатора»? К примеру, медиатор

позволяет каждому модулю функционировать абсолютно независимо от других

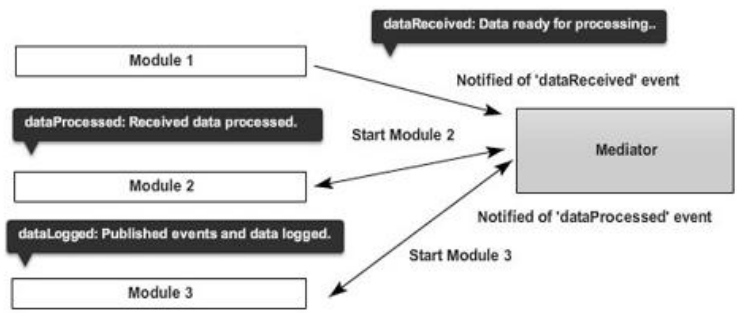
компонентов системы, что приводит к большей гибкости. Если вам ранее уже

приходилось использовать паттерн «наблюдатель» в роли системы доставки

событий между различными частями в вашей системе, то вам не составит труда

разобраться с медиатором.

Давайте посмотрим на модель взаимодействия модулей и медиатора:



Мы можем рассматривать модули, как «издателей», публикующих события.

Медиатор же является и «издателем» и «подписчиком» одновременно. В

примере, Module 1 посылает сообщение, предполагающее некоторую реакцию, медиатору. Затем, медиатор, получив сообщение, уведомляет другие модули об

определенных действиях, которые необходимо выполнить для завершения

задачи. Module 2 выполняет необходимые Module 1 действия, и сообщает о

результате обратно, в медиатор. В это же время, медиатор запускает Module 3

для логгирования поступающих сообщений.

Обратите внимание: здесь нет прямого взаимодействия между модулями. Если

в Module 3 произойдет ошибка или, к примеру, он просто перестанет работать, то медиатор, теоретически, может приостановить выполнение задач в других

модулях, затем перезапустить Module 3 и продолжить работу, практически не

влияя на работу всей системы. Такая слабая связанность модулей является

одним из самых сильных преимуществ паттерна «медиатор», который я вам

предлагаю использовать.

Посмотрим на его преимущества:

Уменьшает связывание модулей, добавляя посредника — центральный элемент

управления. Это позволяет модулям отправлять и слушать сообщения, не

затрагивая остальной части системы. Сообщения могут быть обработаны

любым количеством модулей сразу.

Благодаря слабой связанности кода, внедрение новой функциональности

происходит существенно легче.

И недостатки:

Модули больше не могут взаимодействовать напрямую.

Использование

медиатора приводит к небольшому падению производительности — такова

природа слабой связанности — становится достаточно трудно определить

реакцию системы, отталкиваясь только от событий, происходящих в ней.

Наконец, системы с высокой связанностью кода являются обычно источником

всевозможных проблем, решением которых может стать уменьшение

связанности.

Пример: одна из возможных реализаций паттерна «медиатор», основанная

на работе [@rpflorence]10-8

```
var mediator = (function() {

var subscribe = function(channel, fn) {

if (!mediator.channels[channel]) mediator.channels[channel] = [];
mediator.channels[channel].push({ context: this, callback: fn });
return this;

},


publish = function(channel) {

if (!mediator.channels[channel]) return false;

var args = Array.prototype.slice.call(arguments, 1); for (var i = 0, l
= mediator.channels[channel].length; i < l; i++) {

var subscription = mediator.channels[channel][i];

subscription.callback.apply(subscription.context, args);

}
```

```
return this;
};

return {
channels: {},
publish: publish,
subscribe: subscribe,
installTo: function(obj) {
obj.subscribe = subscribe;
obj.publish = publish;
}
};
}());
```

И два примера использования реализации, написанной выше:

```
//Pub/sub on a centralized mediator
mediator.name = "tim";
mediator.subscribe('nameChange', function(arg) {
console.log(this.name);
this.name = arg;
console.log(this.name);
});
mediator.publish('nameChange', 'david'); //tim, david
//Pub/sub via third party mediator
var obj = {name: 'sam'};
mediator.installTo(obj);
```

obj.subscribe('nameChange', function(arg) {

console.log(this.name);

this.name = arg;

console.log(this.name);

});

obj.publish('nameChange', 'john'); //sam, john

Ссылки по теме:

Stoyan Stefanov — Page 168, JavaScript Patterns

HB Stone — Шаблоны проектирования JavaScript: Медиатор

NVince Huston — Шаблон Медиатора (не относится JavaScript, но кратко)

Использование фасада: абстракция ядра

Фасад, в нашей архитектуре, выполняет роль абстракции ядра приложения.

Фасад находится между медиатором и модулями. Модули в идеальной

ситуации должны взаимодействовать исключительно с фасадом и не должны

знать абсолютно ничего о других компонентах нашей системы.

Фасад также обеспечивает последовательный и доступный в любой момент

интерфейс для модулей. Это похоже на песочницу в архитектуре

Николаса Закаса.

Все сообщения от модулей, адресованные медиатору, проходят через фасад, поэтому он должен быть весьма надежен. Его роль в этом взаимодействии —

анализ сообщений, исходящих от модулей, и передача этих сообщений в

медиатор.

В дополнение к предоставлению интерфейса, фасад также выступает в роли

защиты, определяя с какими частями системы может взаимодействовать модуль.

Модули могут вызывать только свои собственные методы. Для доступа к

другим компонентам системы модулям необходимо иметь специальное

разрешение.

К

примеру,

модуль

может

отправить

сообщение

dataValidationCompletedWriteToDB. В подобных случаях задача фасада —

убедиться, действительно ли этот модуль имеет права на запись в базу данных.

Таким образом, мы пытаемся предотвратить проблемы с модулями, которые

пытаются делать то, что они не должны.

Подведем итоги: медиатор представляет собой интерпретацию паттерна

«подписчик/издатель», но он получает только те сообщения, которые нас

действительно интересуют. За фильтрацию же всех сообщений отвечает фасад.

Использование медиатора: ядро приложения

В этой главе мы кратко пройдемся по некоторым зонам ответственности

медиатора, который играет роль ядра приложения. Но, для начала, давайте

разберемся, что он представляет собой в целом.

Основная задача ядра — управлять жизненным циклом модулей. Когда ядро

получает интересное сообщение от модулей, оно должно определить, как

приложение должно на это отреагировать, таким образом ядро определяет

момент запуска или остановки определенного модуля или набора модулей.

В

идеальной

ситуации,

однажды

запущенный

модуль,

должен

функционировать самостоятельно. В задачи ядра не входит принятие решений

о том, как реагировать, например, на событие DOM ready — в нашей

архитектуре у модулей есть достаточно возможностей для того, чтобы

принимать такие решения самостоятельно.

Возможно, у вас вызовет удивление то обстоятельство, что модулям может

потребоваться остановка. Такое может произойти случае, если модуль вышел

из строя, либо если в работе модуля происходят серьезные ошибки. Решение об

остановке модуля может помочь предотвратить некорректную работу его

методов. Последующий перезапуск таких модулей должен помочь решить

возникшие проблемы. Цель здесь в минимизации негативных последствий для

пользователя нашего приложения.

В дополнение, ядро должно быть в состоянии добавлять или удалять модули

не ломая ничего. Типичный пример — определенный набор функций может

быть недоступен на начальном этапе загрузки страницы, но эти функции могут

быть загружены динамически, на основе определенных действий со стороны

пользователя. Возвращаясь к нашему примеру с GMail, google может, по

умолчанию, держать чат в свернутом состоянии и загружать соответствующий

ему модуль динамически, только в том случае, если пользователь проявит

интерес к использованию этой части приложения. С точки зрения оптимизации

производительности, это должно дать заметный эффект.

Обслуживание ошибок должно также обрабатываться ядром

приложения. В

добавок к сообщению об интересных событиях модули также должны сообщать

и о любых ошибках которые произошли в их работе. Ядро должно

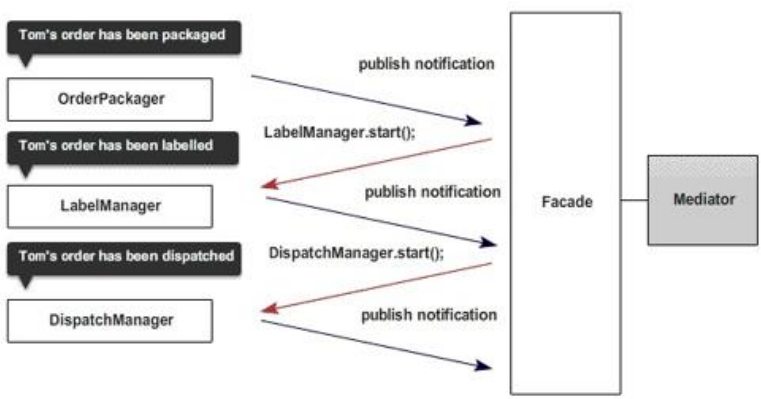
соответствующим образом реагировать на эти ошибки (к примеру, останавливать модули, перезапускать их и тд). Это важно, как часть слабо

связанной архитектуры, позволяющей реализовать новый или лучший подход к

реализации уведомления пользователя об ошибках без ручного изменения в

каждом модуле. Используя механизм для публикации событий и подписки на

них в медиаторе мы сможем достичь этого.



Собираем всех вместе

Модули содержат специфичные части функциональности вашего

приложения. Они публикуют уведомления, информирующие приложение

о том, что случилось что-то интересное. Это их главная забота. Как я

поясню в FAQ, модули могут зависеть от различных

вспомогательных

методов для работы с DOM, но идеальным бы было отсутствие любых

зависимостей от других компонентов системы. Модули не должны иметь

отношение к тому:

какие объекты или модули подписаны на их сообщения, где находятся эти объекты (на клиенте или на сервере), какое количество объектов подписано на уведомления.

Фасад — абстракция ядра защищающая его от прямого контакта с

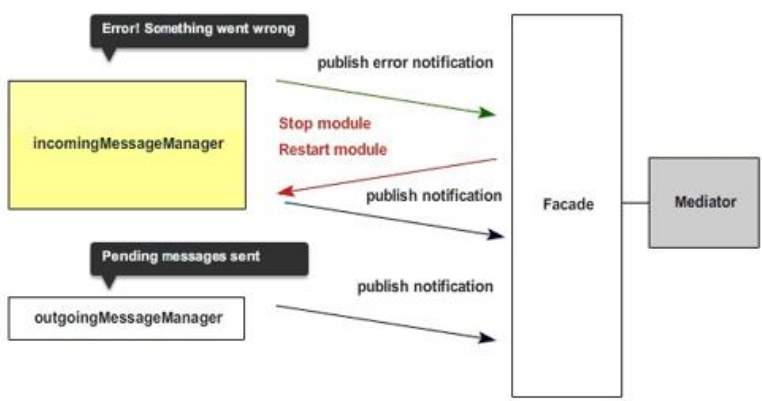
модулями. Он подписывается на интересные сообщения от модулей, и

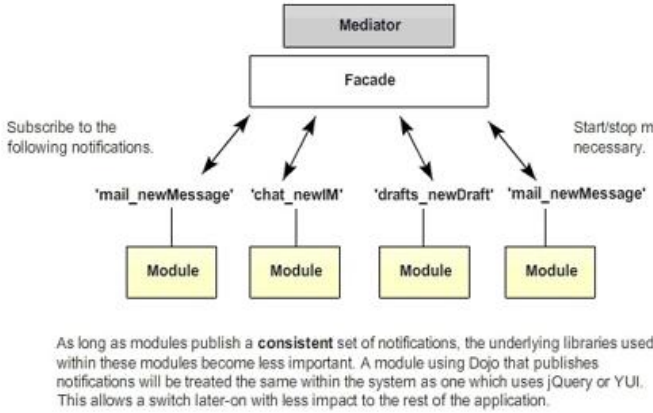
говорит: «Отлично! Что случилось? Расскажи мне больше подробностей!».

Так же фасад поддерживает безопасность модулей, проверяя, действительно ли модуль, отправивший сообщение, имеет необходимые

права для того, чтобы его сообщение было соответствующим образом

обработано ядром.





Медиатор (ядро приложения) выступает в роли управляющего

публикациями событий и подписками на них. Он отвечает за управление

запуском и остановку модулей по необходимости. Здесь используется

частичная динамическая загрузка зависимостей, и гарантия того, что

упавшие модули могут быть централизованно перезапущены в случае

проблем.

Итог этой архитектуры в том, что модули, в большинстве случаев, практически

не зависят от других компонентов приложения. Они могут быть легко

тестируемы и легко поддерживаемы в рамках своего кода. Кроме того, благодаря низкому уровню связанности кода, такие модули можно легко

скопировать на новую страницу для использования в другом проекте, не

прилагая дополнительных усилий. Так же, эти модули могут быть загружены

или удалены динамически в процессе работы приложения.

Развитие идей медиатора: автоматическая

регистрация событий

Как раньше упоминал Михаэль Махемофф, размышляя о больших и

масштабируемых JavaScript-приложениях, весьма выгодно использовать в

приложении больше динамических свойств языка. Вы можете прочитать об

этом больше в его заметках на странице Google+, но я хочу подробнее

остановиться на одной особенности — автоматической регистрации событий

(AER).

AER решает проблему связывания подписчиков и издателей путем добавления

паттерна, который автоматически вызывает нужные методы на основе

соглашения об именовании. Для примера, если модуль публикует сообщение

messageUpdate, произойдет автоматический вызов одноименного метода у всех

модулей, которые такой имеют.

Использование этого паттерна подразумевает регистрацию всех компонентов, которые могут подписываться на события, регистрацию всех событий на

которые можно подписаться и, наконец, для каждого метода подписки в вашем

наборе компонентов должно быть событие. Это выглядит очень интересно по

отношению к нашей архитектуре, но так же имеет ряд интересных проблем.

К примеру, при работе динамически, объекты должны регистрировать себя

после создания. Если вас это заинтересовало — посмотрите пост Михаэля об

AER, где он более подробно обсуждает как бороться с проблемами этого

подхода.

Frequently Asked Questions

Возможно ли обойтись без использования фасада (песочницы)?

Хотя архитектура, которую я изложил здесь, использует фасад для обеспечения

безопасности, вполне возможно достичь того же с помощью медиатора —

сообщения могут обрабатываться непосредственно ядром без использования

фасада. Такая упрощенная версия все равно будет обеспечивать необходимо

низкий уровень связывания кода, но при выборе этого варианта, нужно

понимать, насколько вам будет комфортно работать с модулями, которые

взаимодействуют напрямую с ядром.

В книге говорилось о том, что модули не должны иметь любых

зависимостей, касается ли это библиотек (к примеру, jQuery)

Я специально вынес вопрос о зависимостях от других модулей сюда. Когда

некоторые разработчики выбирают подобную архитектуру, этот выбор

подразумевает что они будут использовать определенные абстракции от DOM-

библиотек. К примеру, вы можете использовать вспомогательную утилиту,

которая будет возвращать нужные вам DOM-элементы используя jQuery (или

dojo). Благодаря этому, модули все еще могут удобно работать с DOM, но уже

будут это делать не напрямую, жестко используя конкретную библиотеку.

Существует достаточно много способов, как сделать модули независимыми, но

стоит понимать, что, в обсуждаемой нами архитектуре, идеальные модули не

должны иметь зависимостей.

Вы заметите, что иногда при таком подходе становится гораздо легче взять

законченный модуль из одного проекта и перенести его в другой проект с

небольшими дополнительными усилиями. Должен сказать, я полностью

согласен с тем, что порой намного лучше, когда модули, вместо определенной

части своей функциональности, просто используют другие модули. Как бы то

ни было, держите в голове то, что такие модули, в некоторых случаях, могут

потребовать гораздо больше усилий для переноса в другой проект.

Я хочу сегодня же начать использовать эту архитектуру. Есть ли

какой-то шаблон от которого я бы мог оттолкнуться?

Когда у меня будет немного свободного времени, я планирую написать для

этой книги бесплатный шаблон проекта, но сейчас, наверное, лучший выбор —

платное учебное пособие написанное Эндрю Бэджис — «Написание

модульного JavaScript» (разоблачу себя: деньги от этой реферальной ссылки, как и любые другие, полученные от этой книги деньги уже инвестируются в

обзор будущих материалов перед тем, как я порекомендую их другим). Пособие

Эндрю включает в себя скринкаст и примеры кода. Оно охватывает большую

часть идей, которые мы обсуждали в книге, но в нем, вместо названия «фасад»

используется слово «песочница», как у Николаса Закаса. Так же, здесь есть

обсуждение о том, что работа с DOM-библиотеками, в идеале, должна быть

реализована посредством абстракции. Я говорил об этом в предыдущем

вопросе. Тут Эндрю делает ставку на некоторые интересные шаблоны, обобщающие работу с селекторами DOM, таким образом, в крайнем случае, замена библиотеки может быть выполнена в несколько коротких строк. Я не

говорю, что это лучший или самый правильный подход, но я поступаю именно

так.

Могут ли модули взаимодействовать с ядром напрямую, если это

необходимо?

Как заметил раньше Николас Закас, технически, нет никаких причин, мешающих модулям напрямую обращаться к ядру. Это

скорее относится к

«лучшим практикам». Если вы намерены строго следовать этой архитектуре, то

вы должны также следовать ее правилам. Либо следовать правилам более

простой архитектуры, которая была описана в первом вопросе.

Credits

Спасибо Николасу Закасу за его оригинальную работу в объединении большого

количества концепций существующих на сегодняшний день. Спасибо Андрэ

Хэнссон за технический обзор книги и за отзывы, которые помогли сделать эту

книгу лучше. Спасибо Ребекке Мюрфей, Джастину Майеру, Питеру Мишо, Полу Айришу и Алексу Секстону и всем, чьи материалы относятся к теме

обсуждаемой в книге и являлись источником вдохновения как для меня так и

для других.

Document Outline

- Паттерны для масштабируемых JavaScript-приложений
- Паттерны для масштабируемых JavaScript-приложений
  - Вступление
  - Что из себя представляет «большое» JavaScript приложение?
  - Давайте обсудим вашу существующую архитектуру
  - Думай о будущем
  - Мозговой штурм
  - Теория модулей
  - Паттерн «Модуль»
  - Литеральная нотация объекта
  - CommonJS Модули
  - Паттерн «Фасад»
  - Паттерн «Медиатор»
  - Использование фасада: абстракция ядра
  - Использование медиатора: ядро приложения
  - Собираем всех вместе
  - Развитие идей медиатора: автоматическая регистрация событий
  - Frequently Asked Questions
  - Credits